INTEGRATION DES METHODES FORMELLES DANS LA SPECIFICATION, LA VERIFICATION ET LA VALIDATION DE MODELES DE SIMULATION A EVENEMENTS DISCRETS

Oumar Maiga, Ufuoma Bright Ighoroje, Mamadou Kaba Traoré

9e Conférence Internationale de Modélisation, Optimisation et SIMulation - MOSIM'12
06 au 08 Juin 2012 - Bordeaux - France
« Performance, interopérabilité et sécurité pour le développement durable »

INTEGRATION DES METHODES FORMELLES DANS LA SPECIFICATION, LA VERIFICATION ET LA VALIDATION DE MODELES DE SIMULATION A EVENEMENTS DISCRETS

Oumar Maïga, Ufuoma Bright Ighoroje

University of Bamako, Bamako, Mali,
African University of Science and Technology, Abuja, Nigeria
maigabababa78@yahoo.fr, uighoroje@aust.edu.ng

Mamadou Kaba Traoré

Clermont Université, Université Blaise Pascal,
CNRS, UMR 6158, LIMOS, F-63173, AUBIERE
Clermont-Ferrand II, France
traore@isima.fr

RESUME : *Ce travail présente une méthodologie d'intégration de méthodes formelles dans la spécification et l'analyse formelle des modèles de simulation à événements discrets avec le formalisme DEVS, dont l'implémentation se base sur les outils hétérogènes associés. Cette méthodologie s'appuie sur le formalisme DDML (DEVS Driven Modeling Language) et combine à divers niveaux de la hiérarchie de spécification introduite en théorie de la simulation, des méthodes formelles qui sont complémentaires pour cerner différentes vues (statiques, dynamiques, fonctionnels) d'un système.*

MOTS-CLES : DEVS, DDML, Simulation, Méthodes formelles.

1 INTRODUCTION

L'objectif visé dans ce travail est l'intégration des méthodes formelles dans le processus de développement de modèles de simulation avec le formalisme DEVS (Zeigler et al. 2000), pour permettre la spécification, la vérification et la validation de modèles. Cette intégration permet de spécifier à la fois les aspects statiques, dynamiques et fonctionnels d'un modèle. L'un des avantages de la combinaison de langages de spécification formels est de prendre avantage sur eux et pouvoir utiliser les outils existants au lieu de construire un nouveau langage. Cette intégration se fera en suivant une hiérarchie de spécification et en construisant un morphisme vers une méthode formelle adaptée, à chaque niveau de cette hiérarchie. Cette méthodologie de combinaison est motivée par le fait qu'une seule méthode formelle ne suffit pas pour capturer toutes les propriétés d'un système complexe. Le fait que DEVS ne définit pas une syntaxe précise pour la spécification des comportements des modèles rend difficile la correspondance avec les méthodes formelles ; Nous utilisons un langage graphique appelé DDML (DEVS Driven Modeling Language) (Traoré 2008) pour établir cette passerelle.

La méthodologie axée sur DDML offre une approche graphique, simple et rigoureuse de création de modèles et permet de transformer ces modèles dans les langages formels cibles pour la vérification de propriétés ou la génération des codes de simulation. Un des objectifs est d'intégrer les résultats dans le Framework du projet Simstudio (Traoré 2008).

2 ETAT DE L'ART

Nous présentons ici les différents travaux réalisés dans le domaine de l'intégration des méthodes formelles dans les processus de Modélisation et Simulation (M&S). Nous nous focalisons surtout sur les travaux réalisés dans l'intégration des méthodes formelles dans la spécification et la vérification formelles de modèles DEVS, et ce du fait du caractère unificateur de ce dernier dans la spécification des systèmes à événements discrets. Il est toutefois important de noter que la méthodologie est applicable à tout autre formalisme s'adressant à la même classe de systèmes (réseau de Pétri, automates à états…), voire à une combinaison de ces formalismes, dès lors qu'une hiérarchie de spécification cohérente est construite pour rendre compte de la structure et du comportement des systèmes ciblés.

Dans (Hong et al. 2006) les auteurs proposent une méthode de vérification des modèles DEVS dans l'environnement DEVSim++. L'approche utilisée consiste à spécifier le modèle en DEVS (formalisme opérationnel) et utiliser la logique temporelle (TL, formalisme des assertions) pour spécifier les propriétés et contraintes temporelles du système.

Une méthode basée sur la réduction de l'ensemble des états et des événements est proposée dans (Hwang et al. 2008). Cette méthode consiste à transformer un modèle DEVS en FD-DEVS (Finite and Deterministic DEVS) et représenter le temps de façon abstraite. Le modèle fini obtenu peut être vérifié de manière exhaustive.

Dans (Cristia 2007), il est proposé un exemple de transformation de modèle DEVS en TLA+ (Lamport 2002). Toutefois, la généralisation de cette conversion DEVS-TLA+ n'a pas été étudiée.

Une méthodologie basée sur la spécification en Z des modèles DEVS et l'utilisation de l'outil Z/EVES (Saaltink 2003) pour la vérification des propriétés a été proposée dans (Traoré 2005) et (Traoré 2006). Une autre

approche d'intégration utilisant les systèmes de transition labélisées et CSP à été proposée dans (Ufuoma et al. 2011). D'autres approches comme (Saadawi et al. 2010) utilisent des restrictions de DEVS pour éviter les problèmes de décidabilité.

La différence de notre approche avec tous ces travaux est que nous proposons une méthodologie basée sur la séparation explicite des différents niveaux d'abstraction. Ceci permet de prendre appui sur l'intégration d'outils existants et efficaces pour divers types d'analyse de propriété. Notre approche est basée sur la hiérarchie de spécification présentée ci-après.

3 HIERACHIE DE SPECIFICATION

La hiérarchie de spécification des systèmes introduite par (Klir 1985) dans un contexte général, a été reformulée dans (Zeigler et al. 2000) dans un contexte plus adapté au domaine de la M&S. Nous réduisons cette hiérarchie aux trois niveaux majeurs auxquels nous considérons que l'activité de M&S se situe (débarrassant ainsi la hiérarchie initiale des niveaux intermédiaires certes significatifs pour la théorie mathématique, mais non concernés par la simulation) : ce sont les niveaux Processus, Système et Traces (voir Figure 1). Nous montrons en section 4 ce que le langage DDML exprime à ces différents niveaux, et nous montrons en section 5, comment construire des morphismes de ces niveaux vers les méthodes formelles.

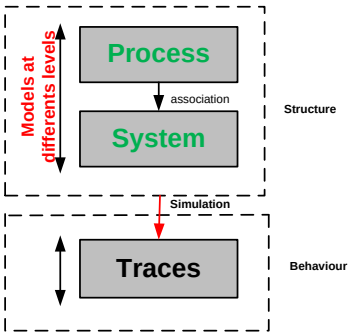

Figure 1 Hiérarchie de spécification

4 DDML

DDML (pour DEVS Driven Modeling Language) est un langage graphique de modélisation et simulation (Traoré 2008) basé sur le formalisme DEVS. Il propose une spécification graphique aux trois niveaux de la hiérarchie précédemment présentée. Les objectifs ayant motivé la définition de DDML sont les suivants :

- Permettre (par son caractère visuel) une plus grande communicabilité des modèles entre différents experts (experts du domaine d'étude et experts de la M&S).
- Avoir une grande puissance d'expression (ce qu'assure DEVS, ce dernier apparaissant comme un dénominateur commun aux langages de simulation à événements discrets (Vangheluwe 2000).
- Autoriser l'accessibilité des modèles à l'analyse formelle et à la synthèse automatique de code (en prenant appui sur des supports logiciels).

4.1 Syntaxe abstraite

La Figure 2 présente le méta modèle donnant la syntaxe abstraite de DDML. Les concepts de base issus de DEVS y ont été enrichis. Nous ne les présentons pas tous, mais seulement les plus fondamentaux.

Un modèle est atomique (*AtomicModel)* ou couplé (*CoupledModel*). Un modèle atomique est la brique de base de la construction des modèles, elle est non décomposable. Un modèle couplé est une agrégation d'autres modèles (atomiques et/ou couplés). Les modèles interagissent entre eux via les interfaces d'entrée-sortie (*IOInterfaces)* appelées aussi ports. Un port est un port d'entrée (*InputPort)* ou un port de sortie (*OutputPort)* en fonction du rôle joué dans les interactions. Les ports servent de canaux de communication pour échanger des événements (*events*) ou des sacs d'événements (*Bags)*.

4.1.1 Niveau Processus

Le niveau processus décrit les modèles couplés. Le modèle couplé et ses composants sont connectés via des matrices de couplage qui sont de trois types : EIC, EOC et IC. La matrice de couplage exterieur-intérieur notée EIC (External Input Coupling) regroupe les connexions entre ports d'entrée du modèle et ports d'entrée de ses composants. La matrice de couplage intérieur-extérieur notée EOC (External Output Coupling) regroupe les connexions entre ports de sortie des composants et ports de sortie du modèle. La matrice de couplage interne notée IC (Internal Coupling) regroupe les connexions entre ports de sortie et ports d'entrée des composants. Une propriété importante au niveau processus est la concurrence entre composants.

4.1.2 Niveau Système

Ce niveau définit la dynamique d'un composant atomique (essentiellement en termes d'états, de transitions d'états et de fonctions de sortie). L'ensemble des états possibles (très souvent infini) est codé par un ensemble (fini) de variables. Un état est obtenu en assignant une valeur à chaque variable. Plusieurs états sont groupables en une configuration, i.e. une caractérisation de contraintes sur les variables d'états. Les transitions sont alors définies entre ces configurations (comme familles de transitions entre états). A chaque configuration est associée une fonction donnant la durée de vie maximale de tout état de cette famille ; cette durée maximale permet de savoir à quel moment une transition (dite interne) doit avoir lieu de l'état courant vers un nouvel état. Si un

événement est reçu avant l'expiration de cette échéance, alors c'est une transition dite externe qui fait évoluer le système vers un nouvel état. Il peut arriver qu'un événement soit reçu exactement au moment où l'échéance expire ; c'est alors une transition de conflit qui est mise en œuvre pour changer d'état. Une fonction de sortie (dite lambda) permet de générer des événements de sortie lors des transitions internes et de conflit.

4.1.3 Niveau Traces

Les traces sont le passé du système en terme d'entrées, de sorties et d'états. Elles établissent donc un historique daté. *AM_FootPrint* est la trace d'un modèle atomique et *CM_FootPrint* est la trace d'un modèle couplé.

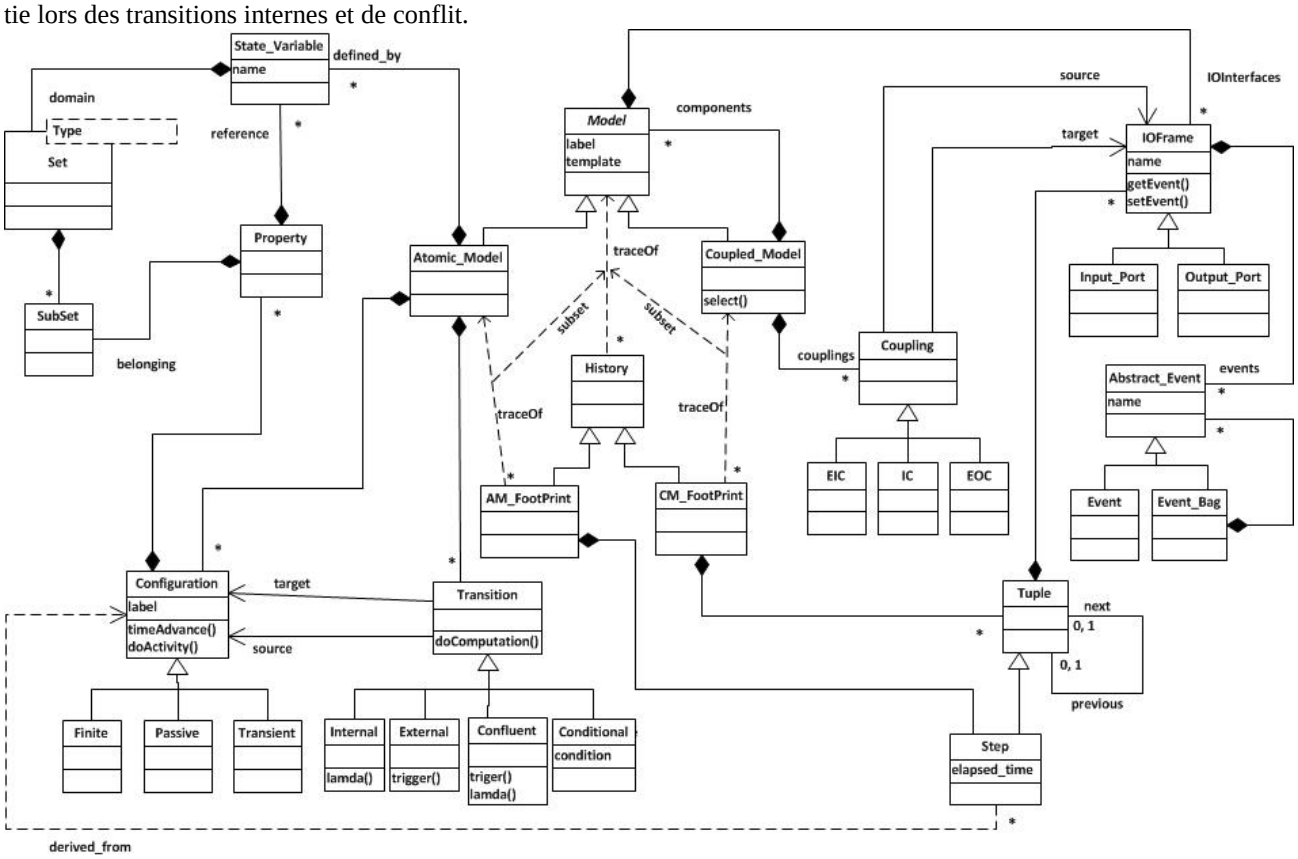

Figure 2. Syntaxe abstraite de DDML

4.2 Syntaxe concrète

Le caractère visuel recherché a nécessité de définir une syntaxe concrète intuitive pour DDML. L'idée fut de s'appuyer sur la puissance de convivialité des classes UML et de les augmenter de symboles et de règles adaptés à la M&S (voir Figure 3). Une classe DDML possède ainsi :

- des attributs (directement déclarés, ou définis par composition, agrégation ou association) ;
- des méthodes (opérations et actions à faire pour changer d'état, faire des traitements annexes dans un état…) ;
- un diagramme de transition de configuration (transitions internes : flèches en traits pleins sur façades verticales, transition externes : flèches en pointillés sur façades horizontales, transitions de conflit : flèches en pointillés particuliers en bout de façades horizontales) où chaque configuration est caractérisée à la fois par sa durée de vie maximale et par les contraintes sur les variables d'état ;
- des ports d'entrée et de sortie (matérialisés par de flèches latérales) dont les domaines de définition peuvent être des types primitifs (entiers, réels…) ou des classes.

Les situations conditionnelles sont exprimées à l'aide d'un losange (test et alternatives multiples). Une distinction visuelle est faite entre configurations à durée de vie nulle (états transcients), configurations à durée de vie infinie (états passifs) et toutes les autres configurations (états finis).

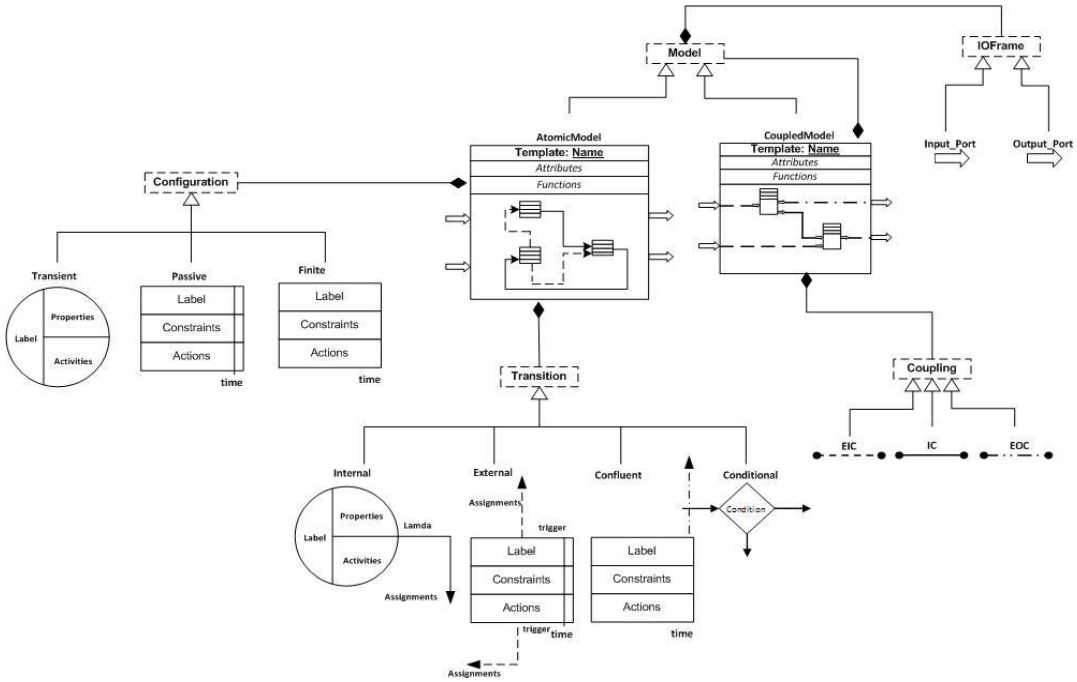

Figure 3. Syntaxe concrète de DDML

5 INTEGRATION DES METHODES FORMELLES EN M&S

5.1 Méthodes formelles

Les méthodes formelles sont des techniques mathématiques utilisables dans toutes les phases de développement : conception, implémentation, tests, maintenance, vérification et validation. Il est possible de les regrouper selon les types de propriétés qu'elles décrivent. Selon (Lamsweerde 2000), on distingue :

- Les méthodes formelles centrées historiques. Elles s'attachent à décrire les propriétés liées aux traces du système. C'est le cas des logiques temporelles, tels LTL, CTL, CTL*, RTL, TLA+...
- Les méthodes formelles centrées états. Elles expriment les états possibles d'un système et les opérations applicables à ces états, opérations qui sont contraintes par des conditions préalables et qui sont sujettes à des conditions postérieures. C'est le cas pour Z (Spivey 1992), VDM (Bjorner et al.1978), Object-Z (Smith 2000), EB (Abrial et al. 2005), LOTOS (Vissers et al. 1989)...
- Les méthodes formelles centrées transitions. Elles proposent une spécification sous forme d'ensemble de transitions d'état. Des exemples de telles approches sont Statecharts (Harel 1998), Promela...
- Les méthodes formelles fonctionnelles. Elles proposent des approches algébriques, telles Larch (Guttag et al. 1993) et ASL (Astesiano et al. 1986) ou des logiques d'ordre supérieur, telles HOL (High Order Logic) et PVS (Prototype Verification System).
- Les méthodes formelles centrées processus. Elles décrivent les systèmes comme des flux de données et de traitement portés par des automates exécutables. C'est le cas des réseaux de Pétri et des algèbres de processus comme CCS (Milner 1985) ou CSP (Hoare 1985).

Plusieurs combinaisons de méthodes formelles ont été proposées dans la littérature. On peut citer les suivantes : ZCCS (Galloway et al. 1997), CSP-OZ (Fischer 2000), CSP2B (Butler 1999), µSZ (Büssow et al. 1997)…

5.2 Méthodologie d'intégration

La spécification d'un modèle au niveau processus définit comment les composants sont interconnectés et comment ils s'influencent. Ce niveau ne nécessite pas la connaissance de ce qui se passe à l'intérieur des modèles qui le composent. Les propriétés mises en avant à ce niveau sont celles que ciblent les méthodes formelles centrées processus (concurrence par exemple).

Au niveau système, la spécification DDML met en évidence la notion d'état et de transition d'état. Les méthodes formelles centrées états ou centrées transitions capturent parfaitement ces propriétés.

Les propriétés exprimées au niveau traces sont celles formellement descriptibles par les méthodes centrées historiques. Nous utiliserons la classe particulière des logiques temporelles.

L'idée est donc d'établir un morphisme entre DDML et une méthode formelle adéquate à chaque niveau de la hiérarchie de spécification comme l'illustre la Figure 4.
Il est important de noter qu'un critère majeur de sélection d'une méthode formelle à un niveau de spécification donné, est l'existence d'outil(s) logiciel(s) support(s) de cette méthode

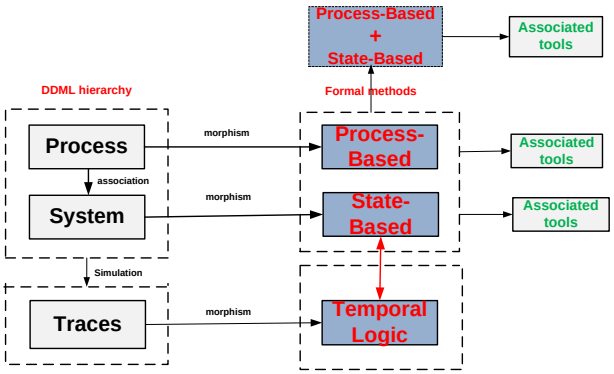

Figure 4. Morphismes DDML – Méthodes formelles

6 EXEMPLE D'INTEGRATION

6.1 Intégration DDML, CSP, Z et CTL

Nous étudions dans cette partie la combinaison de DDML avec Z au niveau système, CSP au niveau processus et CTL au niveau traces pour illustrer la méthodologie présentée précédemment. Nous nous focalisons surtout sur le niveau système dans une étude de cas. A la place de Z, il est possible d'utiliser d'autres méthodes formelles centrées états (ou même centrées transitions). De la même façon il est possible de faire d'autres choix à la place de CSP et CTL. Notre choix est motivé par l'expressivité et la popularité de ces méthodes, ainsi que par la disponibilité d'outils logiciels supports.

Un choix judicieux des langages permet une combinaison sémantiquement bien définie, permettant ensuite de pouvoir faire des vérifications sur la spécification globale. Par exemple pour un choix utilisant Z et CSP, la combinaison CSPZ (Fischer 2000) permet de faire la vérification de la spécification de la structure globale, et TLZ permet de faire la vérification combinée de la dynamique et des traces.

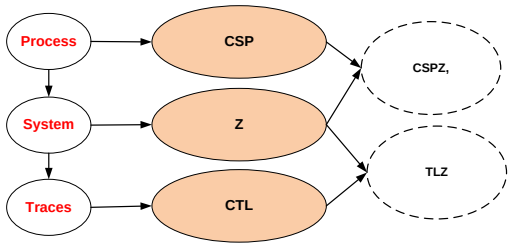

Figure 5. Morphismes de DDML vers Z-CSP-CTL

6.2 Etude de cas

Le protocole du bit alterné (Milner 1980), est un protocole de communication servant à la transmission fiable de messages. Les perturbations peuvent occasionner une perte, altération ou duplication de messages transmis. Dans ce cas, seules les transmissions faites avec succès sont prises en compte, quand un accusé de réception est envoyé à l'adresse de départ du message. Le protocole de communication est composé des composants suivants : envoyeur (Sender), un récepteur (Receiver), deux canaux de transmission (Medium) ; un canal pour la transmission du message et un autre pour l'émission de l'accusé de réception.

Le composant Sender est un modèle atomique DDML (Figure 6).

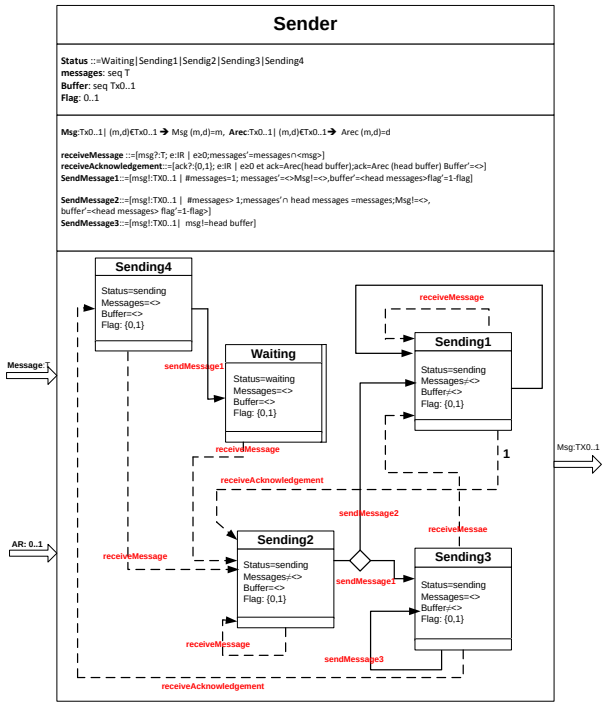

Figure 6. Le composant Sender

Le modèle à deux ports d'entrée; un port pour l'acceptation des messages (le port Message) et un autre pour les accusés de réception (le port AR). T est le type de donnée transmise. Il possède un port pour la transmission des messages reçus avec leurs drapeaux (le port Send). Il procède au retrait d'un nouveau message à transmettre lorsque la file de messages n'est pas vide (c'est-à-dire lorsqu'il se trouve dans la configuration **Sending1**). C'est l'opération **SendMessage** qui définit le traitement de la transmission du message. Tant qu'il ne reçoit pas d'accusé de réception valide et que le buffer (la mémoire contenant le message courant à transmettre) est non vide, il retransmet le message avec le même drapeau (opération **SendMessage3**). S'il n'a plus de messages à transmettre (c'est-à-dire lorsque la mémoire est vide), il passe en attente (configuration **Waiting**). Pendant l'envoi de son message il accepte les autres

messages en les enregistrant dans la file d'attente. C'est l'opération **ReceiveMessage** qui définit le traitement d'un message reçu. S'il reçoit un accusé de réception correct (la réception d'un accusé de réception est définie par l'opération **ReceiveAcknowlegment**), il passe en attente d'un nouveau message (si la file d'attente est vide) ou retire un message de la file d'attente pour transmission (opération **SendMessage1**).

Le composant Receiver est également un modèle atomique (Figure 7) : Il reçoit les messages avec drapeau transmis par Sender via le média de communication. A la réception (opération **ReceiveMessage**) d'un message, il passe à la configuration sendingM si le drapeau du message reçu correspond à son propre drapeau. Il envoi le message (opération **SendMessage**) et un accusé de réception égal à l'inverse de son drapeau (opération **SendAcknowledgment**). Il envoi répétitivement un accusé de réception jusqu'à la réception d'un message avec drapeau correct. La variable r et la fonction f permettent d'introduire un certain degré de probabilité dans le comportement de la ligne.

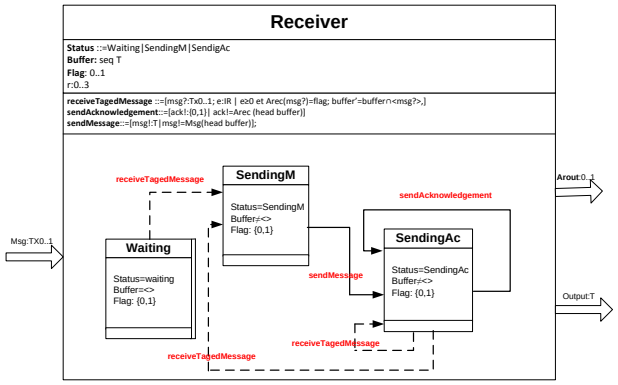

Figure 7. Le composant Receiver

Le canal de transmission est aussi un modèle atomique (Figure 8).

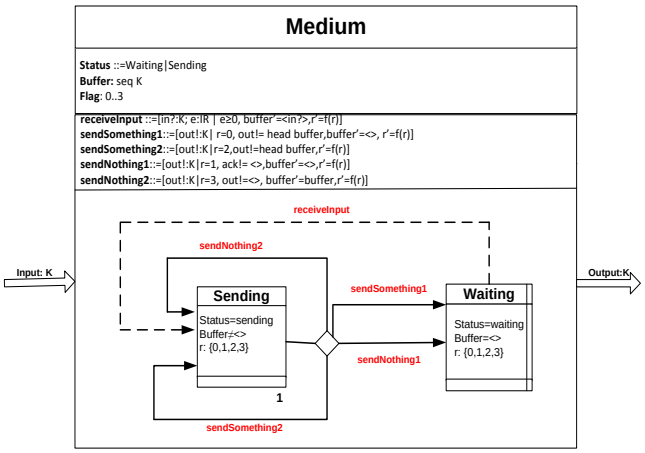

Figure 8. Le canal de communication

Le média supporte des messages de type K. Il transmet le même type de messages sur son seul port de sortie. Initialement, la ligne attend un message (configuration **Waiting**). A la réception d'un message (Opération **receiveMessage**), il passe à la configuration **Sending** pour la transmission du message (**SendSomething1**). Il transmet le message reçu et passe à la configuration Waiting dans lequel, il est prêt à accepter le prochain message. Il est possible qu'il perde le message et dans ce cas il est prêt à accepter le prochain message (**SendNothing1**). . Il peut aussi retransmettre (**SendSomething2**) un message déjà transmis, ou encore prendre du retard dans la transmission (**SendNothing2**).

Le protocole est un modèle couplé (Figure 9). Ses composants sont : Sender, Receiver, le média par lequel Sender envoie des messages et le média via lequel il reçoit des accusés de réception. La priorité entre ces composants est définie par la liste de priorité par défaut. Les modèles canalAccusé et canalMessage sont de type Medium.

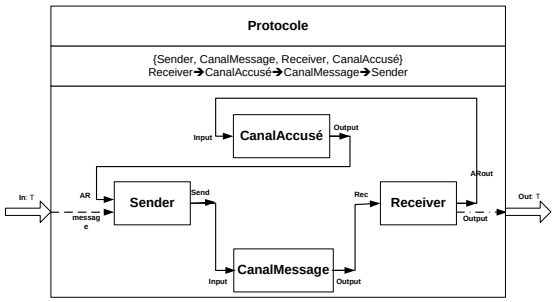

Figure 9. Le protocole Complet

6.3 Transformation du niveau système en Z

Dans la plupart des études de cas de spécifications de systèmes en Z, la méthodologie utilisée consiste à considérer le système comme ayant un schéma d'état global. Les opérations sont définies en termes de changements d'états sur le schéma d'état global résultant de la composition des différents schémas d'état de ses composants. Nous suivrons une approche plus structurée pour décrire les différentes parties du modèle. Nous introduisons la notion de référence d'objet selon la méthode utilisée dans (Hall 1990) pour structurer nos spécifications suivant un style faiblement orienté objet. L'utilisation des références aux objets nous permet d'utiliser des instances de schémas «classes » élémentaires pour construire des modèles et pallier ainsi au problème d'identité des objets dans le Z standard.

Comme un modèle est composé d'un ensemble de ports, de variables d'états, il est nécessaire d'introduire des schémas élémentaires permettant de caractériser ces différents éléments.

Un port est caractérisé par son nom, son domaine de valeur et le modèle associé. Pour distinguer les ports d'entrée et les ports de sortie nous allons introduire une variable qui permet de préciser leurs rôles de port :

$$PortROLE ::= INPUT|OUTPUT$$

$ROLE = INPUT$ pour les ports d'entrée et,
$ROLE = OUTPUT$ pour les ports de sortie.

Nous associons une référence unique à chaque port pour permettre de les manipuler comme des objets avec les caractéristiques de PORT et dont les identités ne changent pas même si les valeurs des attributs changent. Cette notion de référence sera introduite dans tous les schémas élémentaires pour permettre la création de modèles complexes en instanciant des schémas existants.

Soit [$PORT\_REFERENCE$] l'ensemble des références de port. Un port avec une référence unique sera une instance du schéma PORT suivant :

$$
\begin{array}{|l}
\textit{PORT} \\
\hline
Self : PORT\_REFERENCE \\
name : NAME \\
value : DEVS\_Type \\
Model : MODEL \\
Role : PortROLE \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{PORT\_OP} \\
\hline
PORT \\
\hline
self' = self \\
\hline
\end{array}
$$

Pour s'assurer du fait que les identités des ports ne changent pas, quelque soit l'opération, nous définissons une opération PORT_OP qui permet de protéger l'attribut self de toute modification. Il suffit d'inclure ce schéma dans le schéma de toute opération sur un port pour garantir que l'identité de celui-ci ne change pas.

Nous définissons maintenant les opérations permettant des manipuler les ports : modifier le nom d'un port, modifier la valeur d'un port, récupérer le nom d'un port; récupérer la valeur d'un port.

$$
\begin{array}{|l}
\textit{setValuePort} \\
\hline
\Delta(value)Port \\
value? : DEVS\_Type \\
\hline
value : value? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{getValuePort} \\
\hline
\equiv Port \\
value! : DEVS\_Type \\
\hline
value! : value \\
\hline
\end{array}
$$

En particulier, l'opération $setValue$ permet aussi d'initialiser un port avec une valeur initiale. La modification et la récupération du nom d'un port se fait de la même façon.

Nous définissons également une opération permettant de créer un port avec des caractéristiques bien déterminées. L'opération newPort permettant de créer un nouveau port en précisant son nom, son domaine de valeurs, le modèle associé et son rôle, est définie ci-après. La conjonction de l'opération $newPort$ et l'opération $setValue$ permet de créer un port et de l'initialiser.

$$
\begin{array}{|l}
\textit{newPort} \\
\hline
Port \\
name? : NAME \\
type? : \mathbb{P}\ DEVS\_Type \\
Model? : Model \\
Role? : PortRole \\
\hline
name = name? \\
dom(value) = type? \\
Model = Model? \\
Role = Role? \\
\hline
\end{array}
$$

Nous définissons les variables d'état et leurs opérations associées de la même manière que pour les ports. Une variable d'état est caractérisée par sa référence unique, son nom et son domaine de valeurs.

$$
\begin{array}{|l}
\textit{StateVariab} \\
\hline
Self : VAR\_REFERENCE \\
name : NAME \\
value : DEVS\_Type \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{VAR\_OP} \\
\hline
tateVariable \\
\hline
self' = self \\
\hline
\end{array}
$$

Un modèle atomique est une agrégation de ports et de variables d'état muni de fonctions de transition interne et externe, d'une fonction d'avancement du temps et d'une fonction de sortie.

$$
\begin{array}{|l}
\text{Atomic} \\
\hline
self : MODEL\_REFERENCE \\
name : NAME \\
X : seq\ PORT \\
Y : seq\ PORT \\
ROLE : MODEL\_ROLE \\
State : seq\ StateVariable \\
\hline
\forall i : dom(X) \bullet X(i).ROLE = INPUT \\
\forall j : dom(Y) \bullet Y(j).ROLE = OUTPUT \\
ROLE = ATOMIC \\
\hline
\end{array}
$$

Les fonctions (de transition interne, externe, de sortie, et d'avancement du temps) sont définies comme des opérations sur le schéma d'état Atomic de la spécification.

La transformation automatique du modèle atomique DDML se fera en utilisant la syntaxe abstraite de DDML et la syntaxe abstraite XML de Z proposée par le CZT (Community Z Tools, http://czt.sourceforge.net). L'intégration des outils de vérification se fera en exploitant le format XML du modèle DDML. Les détails de la transformation ne sont pas présentés dans ce travail.

6.4 Transformation des niveaux processus et trace

Nous utilisons CSP pour capturer les différents aspects du niveau processus tels que la concurrence en utilisant la technique de transformation ATL conformément au méta modèle UML de CSP. Un modèle couplé est considéré comme un ensemble de processus CSP qui s'exécutent de façon concurrente. L'utilisation des transformations offrent la possibilité de faire de la vérification exhaustive de propriétés écrite en CTL en utilisant des outils comme SPIN, SMV... Pour les traces, nous utilisons CTL de manière similaire. Nous ne donnons pas ici les détails sur cette partie.

7 CONCLUSION ET PERSPECTIVES

Nous avons proposé dans ce travail, une méthodologie d'intégration des méthodes formelles dans le processus de spécification, vérification et validation des modèles

de simulation à événements discrets. Nous avons présenté une instanciation de la méthodologie en utilisant Z, CSP et CTL. La prochaine étape de ce travail est de réaliser l'intégration des outils existants pour ces méthodes formelles à la plateforme SimStudio (Traoré 2008), framework générique, extensible et collaboratif de modélisation, simulation, analyse, et visualisation de modèles DDML. Il propose des moyens de transformation de modèles et de génération de code. Il permet également de faciliter le couplage entre des modèles hétérogènes. La transformation de modèles et la génération de code basées sur une représentation standard XML (Traoré et al. 2010) permet de faciliter l'obtention de solution opérationnelle. L'éditeur DDML (Ufuoma et al. 2011) est utilisé comme plug-in dans le Framework.

Il est possible d'étudier d'autres combinaisons comme Object-Z et CSP ou encore B et CSP. Chaque combinaison présente des avantages et des inconvénients qu'il faudra mettre en évidence dans des travaux à venir.

REFERENCES

Abrial J. R., Métayer C.,Voisin L. 2005. Event-B Language. Deliverable 3.2, 2005.

Astesiano E., Wirsing M., 1986. "An introduction to ASL", *Proc. IFIP WG2.1 Conf. on Program Specifications and Transformations*, North-Holland.

Butler M. 1999. CSP2B: A practical approach to combining CSP and B, *In: Proc. FM'99, Toulouse, France, 22- 24th Sept 1999) 223–241.*

Büssow R., Geisler R., Grieskamp W., Klar M. 1997. The µSZ Notation. *Version 1. Technical Report, TU Berlin, Germany.*

Cristia M. 2007. A TLA+ encoding of DEVS models. *International M&S Multiconference*, Buenos Aires (Argentina), pp. 17–22.

Bjorner D., Jones C.B. 1978. The Vienna Developpement Method: *The Meta-Language, Lecture Notes in Computer Sciences. Springer. ISBN 3-540-08766-4.*

Fischer C. 2000, Combination and Implementation of Processes and Data : from CSP-OZ to Java. *PhD thesis, University of Oldenburg, 2000.*

Galloway A.J., Stoddart W.J. 1997. An Operational Semantic for ZCCS. *In 1st IEEE International Conf. on Formal Engineering Methods, Hiroshima, Japon, 12-14 November 1997.*

Guttag J.V., Horning J.J. 1993. *LARCH: Languages and Tools for Formal Specification*, Springer-Verlag.

Hall A. 1990. Using Z as a Specification Calculus for Object-Oriented Systems, in VDM&Z- Formal Methods in Software Development. *D. Bjorner, C.A.R. Hoare, and H. Langmaack, Eds. 1990, pp. 290-318, Springer-Verlag.*

Harel D. 1998, Modeling Reactive Systems with Statecharts, *Mc Graw Hill (1998).*

Hoare C.A.R. Communicating Sequential Processes. Prentice *Hall (1985).*

Hong J., Song H., Kim T., Park K. 1997. A Real-time discrete-event system specification formalism for seamless real-time software development. *Discrete Event Systems: Theory and Applications*, vol. 7, pp. 355–375.

Hwang M.H., Zeigler B.P. 2009. Reachability Graph of Finite and Deterministic DEVS Networks. *IEEE Trans. on Automation Science And Engineering*, 6 (3).

Jahanan F., Mok A.K. 1994. Modechart: A Specification Language for Real-Time Systems. *IEEE Trans. on Software Engineering, 20 (12), December 1994.*

Klir G.J. Architecture of Systems Complexity. Sauders, New York, 1985.

Lamport L. 2002. Specifying systems: The TLA+ Language and Tools for HardwareSand software Engineers. *Addison-wesley Professionnal, 2002.*

Lamsweerde A.V. 2000. Formal specification: a rodmap. *ICSE Proc. of the Conf. on the future of software engineering, ACM, New York, USA 2000.*

McMillan K.L. 1992, Symbolic Model Checking: An approach to the state explosion problem, *Ph.D. Thesis, Carnegie Mellon University, 1992.*

Milner R. 1980. A Calculus of Communicating Systems, *Springer Verlag (1980).*

Pnueli A. 1981. The temporal semantics of concurrent programs, *Theoretical Computer Science, volume 13 (1981) 45–60.*

Saadawi, H., Wainer G. 2010. From DEVS to RTA-DEVS. *IEEE/ACM 14th Intern. Symp. on Distrib. Simulation & Real Time Applications*, 2010 pp. 207-210.

Saaltink M. 1999, The Z/EVES User's Guide, *Technical Report TR-97-5493-06, ORA Canada (October 1999).*

Smith, G. The Object-Z Specification Language. *Advances in Formal Methods. Kluwer Academic Publishers. 2000.*

Spivey K. 1992. Understanding Z: a specification language and its formal semantics, *Cambridge tracts in Theoretical Computer Science (1992).*

Traoré M. K. Making DEVS Models Amenable to Formal Analysis, *in: Proc. SpringSim, Huntsville, Alabama, USA, April 2-6 2006) 33-39.*

Traoré M. K. 2009. A Graphical Notation for DEVS, *in: Proc. SpringSim, San Diego, CA, USA, 2009.*

Ufuoma B.I., Maïga O., Traoré M.K. A Formal Framework For the DEVS Driven Modeling Language. *Proc. EMSS 2011.*

Ufuoma B.I., Traoré M.K. A Graphical Editor For the DEVS Driven Modeling Language. *Proc. ESM 2011.*

Vangheluwe H. 2000. DEVS as a Common Denominator for Multi-*Formalism* Hybrid Systems Modeling, *in: Proc. IEEE Intern. Symp. on Computer-Aided Control System Design, 129-134.*

Vissers C.A., Van P., Eijk H.J., Diaz M. 1989. The Formal Description Technique LOTOS. *Elsevier Science Publishers, 1989.*

Zeigler, B., Praehofer, H., Kim T. *Theory of Modeling and Simulation*. 2nd Edition. Academic Press, 2000.